

Efficient repeat finding via suffix arrays

Verónica Becher Alejandro Deymonnaz Pablo Ariel Heiber
vbecher@dc.uba.ar adeymo@dc.uba.ar pheiber@dc.uba.ar

Departamento de Computación, Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires
Argentina

April 2012

Abstract

We solve the problem of finding interspersed maximal repeats using a suffix array construction. As it is well known, all the functionality of suffix trees can be handled by suffix arrays, gaining practicality. Our solution improves the suffix tree based approaches for the repeat finding problem, being particularly well suited for very large inputs. We prove the correctness and complexity of the algorithms.

Keywords: repeats, suffix array, longest maximal substring.

1 Introduction

Many basic problems on strings have been solved in the past using the *suffix tree* data structure because of its theoretical linear complexity bounds, [4]. However, suffix trees proved to be impractical when handling very large inputs, as needed in comparative genomics or web indexing. The linear time and memory bounds of suffix trees hide a large constant factor, so many of the algorithms based on suffix trees were superseded by algorithms based on suffix arrays, see [7, 14, 18].

In this paper we focus on the classical problem of finding repeats in a given string, and we give a solution based on the *suffix array*. This is the kind of algorithm that everyone believes it must have already been done. Indeed, we published our algorithm without a formal analysis in 2009 in [1]. The algorithm has been recently cited as the state of the art for repeat finding with suffix arrays [10]. The purpose of the present paper is to document the correctness of the algorithm and to give an analysis of its time and memory complexity. In addition, the version of the algorithms we give here contain some improvements in memory usage.

We consider two variants of the repeat finding problem. The first one is to find all *maximal repeats* in a given string, where a maximal repeat is a substring that occurs at least twice, and all its extensions occur fewer times. The second is the problem of finding all the maximal substrings in a give string that occur more than once. Using the terminology of Gusfield [4], we call them *supermaximal* prepeats.

2 Notation and preliminary definitions

Notation. Assume the alphabet \mathcal{A} , a finite set of symbols. A string is a finite sequence of symbols in \mathcal{A} . The length of a string w , denoted by $|w|$, is the number of symbols in w . We address the positions of a string w by counting from 1 to $|w|$. The symbol in position i is denoted $w[i]$, and $w[i..j]$ represents the substring that starts in position i and ends in position j of w , inclusive. A prefix of a string w is an initial segment of w , $w[1..\ell]$. A suffix of a string w is a final segment of w , $w[i..|w|]$. We say u is a substring of w if $u = w[i..j]$ for some i, j . If u is a substring of w we say that u occurs in w at position i if $u = w[i..i + |u| - 1]$. When u is a substring of w , we say that w is an extension of u .

Definition 1 (Maximal repeat, [4]). *A maximal repeat of a string w is a string that occurs more than once in w , and each of its extensions occurs fewer times.*

Example 2. The set of maximal repeats of $w = abcdeabcfbcde$ is $\{abcd, bcde, bcd\}$. Clearly $abcd$ and $bcde$ are maximal repeats occurring twice. But also bcd is a maximal repeat because it occurs three times in w , and every extension of bcd occurs fewer times. There are no other maximal repeats in w (bc , for example, occurs three times, but since bcd occurs the same number of times, bc is not a maximal repeat.)

This example already shows that maximal repeats can be nested and overlapping. A bound of the number of maximal repeats on a string is already given in [4].

Theorem 3 ([4], Theorem 7.12.1). *The number of maximal repeats in a string of length n is no greater than n .*

This result can also be derived from Algorithm 1 and Theorem 11.

3 An algorithm to find all maximal repeats

Let w be a string of length $n = |w|$. The suffix array ([16]) of w is a permutation r of the indices $1..n$ such that for each $i < j$, $w[r[i]..n]$ is lexicographically less than $w[r[j]..n]$. Thus, a suffix array represents the lexicographic order of all suffixes of the input w . For convenience we also store the inverse permutation of r and call it p , namely, $p[r[i]] = i$. As a first step of our procedure we use the fast algorithm of [14] to build the suffix array of the input w in time $\mathcal{O}(n \log n)$.

We can think each substring of w as a prefix of a suffix of w . Suppose a maximal repeat u occurs k times in w ; then, it is a prefix of k different suffixes of w . Since the suffix array r records the lexicographical order of the suffixes of w , the maximal repeat u can be seen as a string of length $|u|$ addressed by k consecutive indices of r . Namely, there will be an index i such that u occurs at positions $r[i], r[i + 1], \dots$, and $r[i + k - 1]$ of w . The algorithm has to identify which strings addressed by consecutive indices of the suffix array are indeed maximal repeats.

We compute the length of the longest common prefix of each pair of consecutive suffixes in the lexicographic order. For this task we use the linear time algorithm of Kasai et al. [8]. For any position $1 \leq i < n$, $LCP[i]$ gives the length of the longest common prefix of $w[r[i]..n]$ and $w[r[i + 1]..n]$.

Definition 4. *A substring of a given string is maximal to the right when all its extensions to the right occur fewer times; similarly for maximality to the left.*

This definition allows for a characterization of maximal repeats that we use in Algorithm 1.

Proposition 5. *A substring of a given string is a maximal repeat if and only if it is maximal to the left and to the right.*

To identify maximal repeats we first identify the candidates that are maximal to the right, and then we filter out those that are not maximal to the left. The next propositions assume the suffix array r of the input string w , its inverse permutation p , and the LCP array.

Proposition 6. *A substring u of w is repeated and maximal to the right if and only if there is an index i , $1 \leq i < n$, and a number of occurrences $k \geq 2$ such that*

1. $u = w[r[i]..r[i] + |u| - 1]$;
2. u occurs exactly k times in w ,
 - for each $t \in [i, i + k - 2]$, $LCP[t] \geq |u|$,
 - $LCP[i - 1] < |u|$ or $i = 1$,
 - $LCP[i + k - 1] < |u|$ or $i + k - 1 = n$,
3. There is $t \in [i, i + k - 2]$ such that $LCP[t] = |u|$.

Proof. We first prove the implication to the right. Assume u occurs more than once in w , and let k be its number of occurrences. Let $i = \min\{t : w[r[t]..r[t] + |u| - 1] = u\}$ and $j = \max\{t : w[r[t]..r[t] + |u| - 1] = u\}$. By definition of r , for each t such that $i \leq t \leq j$, $w[r[i]..n]$ is lexicographically before $w[r[t]..n]$ which in turn is lexicographically before $w[r[j]..n]$. Since the first $|u|$ characters of the suffixes addressed by i and j are the same, for every such t , $w[r[t]..r[t] + |u| - 1] = u$. Therefore, the longest common prefix of any two of these strings has length at least $|u|$. Since j is the largest index and i is the smallest, for any other index h outside the range of $i..j$, either $n - r[h] < |u|$ or $w[r[h]..r[h] + |u| - 1] \neq u$, therefore the longest common prefix between $w[r[i]..n]$ and any of the suffixes outside the interval addressed by $i..j$ is necessarily less than $|u|$. Finally, since u is maximal to the right, there must be at least two occurrences of u having different extensions to the right. This causes a value in the LCP to be exactly $|u|$.

The reverse implication is implied by the following observation. Let $u = w[r[i]..r[i] + |u| - 1]$ and assume all strings $w[r[t]..r[t] + |u| - 1]$ with $i \leq t < j = i + k$ share its first $|u|$ characters, so they all start with u . Assume there is some t , $i \leq t < i + k - 1$ such that $LCP[t] = |u|$; therefore, either $n - \max(r[t], r[t + 1]) = |u|$ or $w[r[t] + |u|] \neq w[r[t + 1] + |u|]$. In either case, the repeat cannot be extended to the right. Since the previous and next strings outside the interval addressed by $i..j$, if they exist, have a longest common prefix of length less than $|u|$, they do not start with u . By definition of r , every suffix starting with u has to be included in the considered interval. Hence, every extension of u to the right occurs fewer times than u itself. \square

Proposition 7. *A substring u of w maximal to the right, is also maximal to the left if and only if there is an index i , $1 \leq i < n$, and a number of occurrences $k \geq 2$ such that*

1. u occurs k times in w ,
2. $w[r[i] - 1] \neq w[r[i + k - 1] - 1]$, or
 - $r[i] = 1$, or $r[i + k - 1] = 1$, or
 - $p[r[i + k - 1] - 1] - p[r[i] - 1] \neq k - 1$.

Proof. We use the characterization of maximality to the right given in Proposition 6. Assume u is maximal to the right, let i be its first index in r and let k be its number of occurrences. We first show the implication to the left by proving the contrapositive. Suppose u is not maximal to the left, then there is a symbol c such that cu occurs the same number of times as u . All occurrences of cu also contain u , hence for each of the k occurrences of u there is a c before u . Since u occurs at positions $r[i], \dots, r[i+k-1]$ in w , cu must occur at positions $r[i]-1, \dots, r[i+k-1]-1$. Thus, $r[i] \neq 1$, $r[i+k-1] \neq 1$ and $w[r[i]] = w[r[i+k-1]] = c$. Observe that the relative order in r of the suffixes starting with u is the same as the relative order of all the suffixes starting with cu , because they are the same strings, with a c added at the front. So, for each $j = 0 \dots k-1$, $p[r[i]-1] = p[r[i+j]-1] + j$, and in particular, $p[r[i]-1] = p[r[i+k-1]-1] + k-1$.

For the implication to the right assume u is repeated, maximal to the right and maximal to the left. Since u occurs k times, $k \geq 2$, there is an index i such that for every t , $i \leq t \leq i+k-1$, $u = w[r[t]..r[t]+|u|-1]$. Assume $r[i] \neq 1$, and $r[i+k-1] \neq 1$, and $p[r[i+k-1]-1] - p[r[i]-1] = k-1$. Since u is maximal to the left, there must be at least two positions a and b of w witnessing that u cannot be extended to the left. Either $w[a-1] \neq w[b-1]$ or $a = 1$ or $b = 1$ (recall the positions of a string are numbered starting at 1), while $u = w[a..a+|u|-1] = w[b..b+|u|-1]$. But $a = r[t]$ and $b = r[t']$ for some indices t, t' in the range $i..i+k-1$. Since r records the lexicographic ordering, if that happens on any pair of positions t, t' , $i \leq t < t' \leq i+k-1$, it also happens choosing $t = i$ and $t' = i+k-1$, therefore, if none of those indices is 1, $w[r[i]-1] \neq w[r[i+k-1]-1]$. \square

We call the algorithm **findmaxr**. Its pseudocode is described in Algorithm 1. It takes as an extra parameter an integer ml which is the minimum length of a maximal repeat to be reported (it can be set to 1 if desired).

The idea of the algorithm is to treat all suffix intervals as described in Proposition 6, one by one, but in non-decreasing order of their longest common prefix. We use a set S to keep track of the LCP values already seen at each current step of the algorithm, and the fake indices 0 and n to treat border cases. The algorithm first discards all values less than ml , inserting them into S , hence, the required computational time in practice diminishes as ml increases.

The main loop of the algorithm iterates on the LCP values in non-decreasing order. For each current value, the algorithm finds the largest interval of indices in the suffix array r , above and below it, such that all the contained LCP values are greater than or equal to the current one. Being the largest interval ensures that all occurrences of the addressed string are taken into account. Then the algorithm checks whether this interval implies a string that is maximal to the right and to the left. Since the main loop iterates on LCP values in non-decreasing order, each current interval of indices in r lays between the closest previously used indices above and below it. Notice that the interval is only valid when the limits are strictly less (and not equal) than the current LCP value. Each of the intervals considered in Proposition 6, is addressed in the first visited of all j such that $LCP[j] = |u|$.

The algorithm keeps track of the set S of indices already seen; these are the indices of LCP that have already been treated in the main loop of the algorithm. The data structure to implement this set S should be efficient for the insertion operation in S , as well as the queries for the “minimum greater than a given value” and the “maximum less than a given value”. Notice that these are the most expensive operations of the main loop, being critical for the overall time complexity. All other operations in the main loop are $O(1)$.

Algorithm 1 findmaxr(w, ml)

$n := |w|$
 $r :=$ suffix array of w
 $p :=$ inverse permutation of r
 $LCP :=$ longest common prefix array of w and r
 $S := \{u : LCP[u] < ml\} \cup \{0, n\}$ —discard all indices of w whose LCP is less than ml —
 $I :=$ permutation of $[0, n - 1]$ such that $LCP[I[i]] \leq LCP[I[i + 1]]$
 $initial := \min\{t : LCP[I[t]] \geq ml\}$
for $t := initial$ to $n - 1$ **do**
 $i := I[t]$
 $p_i := \max\{j : j \in S \wedge j < i\} + 1$
 $n_i := \min\{j : j \in S \wedge j > i\}$
 $S := S \cup \{i\}$
 if ($p_i = 1$ or $LCP[p_i - 1] \neq LCP[i]$) and ($n_i = n$ or $LCP[n_i] \neq LCP[i]$) **then**
 —here we have a substring maximal to the right, check if it is maximal to the left—
 if $r[p_i] = 1$ or $r[n_i] = 1$ or $w[r[p_i] - 1] \neq w[r[n_i] - 1]$ or $p[r[n_i] - 1] - p[r[p_i] - 1] \neq n_i - p_i$
 then
 —here it is both maximal to the right and to the left—
 report the maximal repeat $w[r[i]..r[i] + LCP[i] - 1]$ and its $n_i - p_i + 1$ occurrences,
 whose list of positions in w are contiguous in r starting at p_i .
 end if
 end if
 end for

Algorithm 2 Maximum less than(t)

—inquire the bit tree S —
repeat
 if t is a left child **then**
 $t :=$ rightmost node to the left of t in its level
 else
 $t := \text{parent}(t)$
 end if
until node t is set to 1
while t is not a leaf **do**
 if right child(t) is set to 1 **then**
 $t :=$ right child(t)
 else
 $t :=$ left child(t)
 end if
end while
return t

We represent the set S with a small data structure, which is efficient for the mentioned operations ensuring they take time $\mathcal{O}(\log n)$. This data structure requires to have the number of elements of the universe specified in advance. In our case this is not a problem, because the universe is the integer interval $[0, n + 1]$. We implement S as a binary tree of bits with $n + 2$ leaves. Each leaf represents one of the $n + 2$ elements of the universe, the value is 1 if the element is in S , and it is 0 otherwise. An internal node has value 0 if and only if both of its children are 0, otherwise it has value 1. The insertion operation only needs to update the branch of the modified leaf, so it can be done in $\log n$ time.

Algorithm 2 gives the pseudocode to find the maximum less than a given t . The idea is to go up in the tree always moving to the rightmost node that has a chance of being set to 1 because of a leaf to the left of the parameter t ; this rapidly increases the interval represented by the nodes, because each move up multiplies the size of the checked interval by at least $3/2$. As soon as we find a node with value 1, we immediately go down in the tree looking for the rightmost leaf that caused that 1. Finding the minimum greater than t is analogous.

3.1 Improved implementation saving n word_size memory

We give an alternative implementation of Algorithm 1 that lowers the needed memory to n ($3\text{word_size} + \log |\mathcal{A}| + 2$), while keeping the same time complexity, cf. Theorems 13 and 14. In this variant we get rid of the resident LCP array before entering into the main loop, and we efficiently deal with the three LCP values needed inside the mainloop.

Algorithm 3 gives the pseudocode. It starts exactly as Algorithm 1, first computing the suffix array r , the LCP array and the set S . Then it builds the array I using the simple algorithm *Inverse a permutation in place* of [9] Algorithm I, Section 1.3.3, attributed there to [6]. This inversion requires time $\mathcal{O}(n)$ and n auxiliary bits. However, the LCP array is *not* a permutation of $1..n$, so we first map the LCP into a permutation $1..n$, using an auxiliary array. At this point we can free the LCP space, and the array I is constructed in the same place of the auxiliary permutation.

Both, Algorithm 1 and Algorithm 3, in their main loop iterate the indices in non-decreasing order of the LCP values. However, Algorithm 3 requires that *in case of tie of the LCP values the indices be iterated in increasing order*. This ordering is ensured when we construct the array I . The rest of the implementation is exactly as Algorithm 1.

Proposition 8. *The total memory required by Algorithm 3 sums up to the original input w , the set S , and three arrays of length n with values between 0 and n .*

Proof. The input w , the set S , the suffix array r are permanently in memory. Besides there are at most two other integer arrays of length n in memory. First the LCP and the auxiliary permutation array. Then LCP is discarded and array I is constructed in place of the auxiliary permutation using n auxiliary bits. Then, these auxiliary bits are discarded and array p is constructed. \square

The total number of instructions involved in the whole computation of the main loop of Algorithm 3 and those in the main loop of Algorithm 1 differ in $\mathcal{O}(1)$. Therefore, the overall time complexity of the two variants of **findmaxr** is the same.

Proposition 9. *The overall computation of all the needed LCP values in the main loop of Algorithm 3 require at most $\mathcal{O}(n)$ comparisons.*

Algorithm 3 findmaxr(w, ml)

```
 $n := |w|$ 
 $r :=$  suffix array of  $w$ 
 $p :=$  inverse permutation of  $r$ 
 $LCP :=$  longest common prefix array of  $w$  and  $r$ 
 $S := \{u : LCP[u] < ml\} \cup \{0, n\}$  —discard all indices of  $w$  whose  $LCP$  is less than  $ml$ —
 $I :=$  permutation of  $[0, n - 1]$  such that  $LCP[I[i]] \leq LCP[I[i + 1]]$ 
 $initial := \min\{t : LCP[I[t]] \geq ml\}$ 
 $lasti := -1$ 
 $lastLCP := 0$ 
 $curLCP := 0$ 
for  $t := initial$  to  $n - 1$  do
   $i := I[t]$ 
  while  $w[r[i] + curLCP] = w[r[i + 1] + curLCP]$  do
     $curLCP := curLCP + 1$ 
  end while
   $p_i := \max\{j \in S \wedge j < i\} + 1$ 
   $n_i := \min\{j \in S \wedge j > i\}$ 
   $S := S \cup \{i\}$ 
  if  $(p_i = 1 \text{ or } (lasti = p_i - 1 \text{ and } lastLCP \neq curLCP))$  then
    —here we have a substring maximal to the right, check if it is maximal to the left—
    if  $r[p_i] = 1$  or  $r[n_i] = 1$  or  $w[r[p_i] - 1] \neq w[r[n_i] - 1]$  or  $p[r[n_i] - 1] - p[r[p_i] - 1] \neq n_i - p_i$ 
    then
      —here it is both maximal to the right and to the left—
      report  $w[r[i]..r[i] + curLCP - 1]$  and its  $n_i - p_i + 1$  occurrences, whose list of positions
      in  $w$  are contiguous in  $r$  starting at  $p_i$ .
    end if
  end if
   $lastLCP := curLCP$ 
   $lasti := i$ 
end for
```

Proof. The main loop of Algorithm 1 uses three values in the LCP array. In Algorithm 3 we compute the needed values, profiting that indices i are visited in non decreasing order of their LCP values, and those having the same LCP value are visited in increasing order (this ordering is achieved when we construct array I). To compute $LCP[i]$ do the comparison of the two suffixes character by character *but starting from the previously used LCP value*. Hence, the total number of character comparisons in the overall computation of all the LCP values is at most $2n$ (there can be at most n comparisons that yield each of the 2 possible results). For the comparison $LCP[p_i - 1] \neq LCP[i]$: Since $p_i - 1 = \max\{j \in S \wedge j < i\}$, the index $p_i - 1$ was already seen, so its LCP value is no greater than $LCP[i]$. In case $LCP[i]$ differs from the LCP value of the last index seen, given that indexes are visited in non decreasing order of their LCP value, and $p_i - 1 \in S$, hence it was already seen, we conclude $LCP[p_i - 1] \neq LCP[i]$. In case $LCP[i]$ equals the LCP value of the last index seen, since

indexes having the same LCP value are visited in increasing order, $LCP[p_i - 1] = LCP[i]$ exactly when $p_i - 1$ coincides with the last index seen, because $p_i - 1$ is the largest seen index smaller than i . For the comparison $LCP[n_i - 1] \neq LCP[i]$: Since indices with the same LCP value are visited in increasing order, and $n_i = \min\{j \in S \wedge j > i\}$, namely, n_i is the smallest seen index greater than i , then necessarily $LCP[n_i] < LCP[i]$. Thus, the inequality $LCP[n_i] \neq LCP[i]$ is always true. \square

3.2 Correctness of algorithm findmaxr

Theorem 10 (Correctness). *The algorithm **findmaxr** computes all occurrences of all maximal repeats in the input string, in increasing order of length.*

Proof. Consider Algorithm 1. The main loop sequentially access the array I , the permutation array for the non-decreasing order of LCP . By Proposition 5 maximal repeats are exactly the candidates that are maximal to the right and to the left. Propositions 6 and 7, fully characterize these properties with conditions on the data structures LCP , r , and p . The algorithm checks these conditions. \square

3.3 Complexity of algorithm findmaxr

As is custom in the literature on algorithms we express the time and space complexity assuming integer values can be stored in a unit, and integer additions and multiplications can be done in $\mathcal{O}(1)$. These assumptions make sense because the integer values involved in the algorithm fit into the processor word size for practical cases. Although the algorithm is scalable for any input size, the derived complexity bounds are guaranteed only if the input size remains under the machine addressable size. Otherwise, the classical logarithmic complexity charge for each integer operation becomes mandatory.

Assume an input size of n symbols. To bound the time complexity of our main algorithm, we first show that the set of all maximal repeats and their occurrences can be represented in a concise way.

Theorem 11. *For any string w of length n , the set of all maximal repeats and all their occurrences is representable in space $\mathcal{O}(n)$.*

Proof. Each iteration of the main loop of Algorithm 1 reports at most one maximal repeat, followed by the list of all its occurrences. Each reported maximal repeat and all its occurrences can be represented with three unsigned integers: an index i in the suffix array r , a length ℓ , and the number of occurrences m . The reported maximal repeat is the prefix of length ℓ of the suffix at position $r[i]$. Its m occurrences are respectively in positions $r[i], \dots, r[i + m - 1]$. Each of these integers is at most n (where n is at most the maximum addressable memory) and we need n of them. Assuming that these integer values can be stored in fixed number of bits, this output requires size $\mathcal{O}(n)$. Finally, we need to store the suffix array r , which contains n integer values that are a permutation of $1..n$, so it also requires $\mathcal{O}(n)$. The input w also takes $\mathcal{O}(n)$ space, since each symbol in \mathcal{A} also takes $\mathcal{O}(1)$ because $|\mathcal{A}| \leq n$. \square

Of course, if instead of charging a fixed number of bits to store an integer, we count the length of its bit representation, the total needed output space to report all maximal repeats and occurrences in a given input string of length n becomes $\mathcal{O}(n \log n)$. The input w in this

case would have the same $\mathcal{O}(n \log n)$ bound, but probably takes a lot less because alphabet sizes are usually small compared with n .

Maximum_less_than and Minimum_greater_than take $\mathcal{O}(\log n)$ time. We prove one, the other is analogous.

Proposition 12. *The time complexity of Maximum_less_than a given value is $\mathcal{O}(\log n)$.*

Proof. In the repeat loop of Algorithm 2 there is at most one move to the right for each move up in the tree, hence, we have in total $\mathcal{O}(\log n)$ iterations. Then, in the while loop, every move goes down one level, therefore there are also $\mathcal{O}(\log n)$ total moves. If the tree is implemented over a bit array—in our implementation we use a bit array for each level in the tree—, all moves are easily implemented in $\mathcal{O}(1)$; thus, the entire running time of each query is $\mathcal{O}(\log n)$. \square

Theorem 13 (Time Complexity). *The algorithm **findmaxr** takes time $\mathcal{O}(n \log n)$.*

Proof. Consider Algorithm 1 or its variant as Algorithm 3 together with Proposition 9. All steps before the main for loop take $\mathcal{O}(n \log n)$ operations. The main loop iterates n times. The most expensive procedure performed in the loop body is the manipulation of the tree for the set S , which requires at most $\mathcal{O}(\log n)$ operations, cf. Proposition 12. \square

Theorem 14 (Space complexity). *For an input of size n , algorithm **findmaxr** uses $\mathcal{O}(n)$ memory space. More precisely, it uses $n (3 \text{word_size} + \log |\mathcal{A}| + 2) + \mathcal{O}(1)$ bits.*

Proof. Consider Algorithm 3. The whole input w is allocated in memory. Since it contains n symbols, its memory usage is $n \log |\mathcal{A}|$ bits. By Proposition 8 the total amount of memory needed is for the input w , the set S and three integer arrays of length n with values between 0 and n . The described tree for the set S has $2n + 1$ nodes, implemented with an array of $2n + 1$ bits. Then, the exact memory space of all the data structures is $n (3 \text{word_size} + \log |\mathcal{A}| + 2)$ bits. The local variables are counted in the $\mathcal{O}(1)$ term. \square

4 An algorithm to find all supermaximal repeats

We solve here the problem of finding the maximal substrings of the input that are repeated. These are a subset of the maximal repeats (in the sense of Definition 1) in the input string such that none of their extensions is also a maximal repeat, called *supermaximal repeats*. For instance, in Example 2 the set of all maximal repeats in string $w = abcdeabdcdfbcde$ is $\{abcd, bcde, bcd\}$. While the set of supermaximal repeats is just $\{abcd, bcde\}$ since bcd is a substring of $abcd$ (and also of $bcde$).

Definition 15 (Supermaximal repeats, [4]). *A string u is a supermaximal repeat in w if u is a substring that occurs at least twice in w and each extension of u occurs at most once in w .*

Algorithm 4, called **findsmxr**, finds all supermaximal repeats in a given string w . It is similar to the algorithm **findmaxr** of the previous section, but simpler. We define an analogous notion of maximality to the right and to the left for this case.

Definition 16. *A substring u that occurs more than once in w is supermaximal to the right if all of its extensions to the right occurs at most once in w . Supermaximality to the left is defined analogously.*

Algorithm 4 findsmxr(w, ml)

```
 $n := |w|$ 
 $r :=$  suffix array of  $w$ 
 $LCP :=$  longest common prefix array of  $w$  and  $r$ 
 $up := 1$ 
for  $i := 2$  to  $n - 1$  do
  if  $LCP[i] > LCP[i - 1]$  then
     $up := i$  —starting position in  $LCP$  for the set of local maximum values—
  else
    if  $LCP[i] \neq LCP[i - 1] \wedge LCP[i - 1] \geq ml$  then
      —the indices from  $up$  to  $i - 1$  give a local maximum in  $LCP$  of appropriate length—
      if  $\#\{w[r[j] - 1] : up \leq j \leq i - 1 \wedge r[j] > 1\} = i - up + 1$  then
        —check that all previous characters are different—
        report  $i - up + 1$  supermaximal repeats of size  $LCP[up]$  whose
        list of positions in  $w$  are contiguous in  $r$  starting at  $up$ .
      end if
    end if
  end if
end for
```

Proposition 17. *A substring of w is supermaximal to the right if and only if it is addressed by consecutive indices in the suffix array denoting a local maximum in LCP .*

Proof. A substring u of w is supermaximal to the right, cf. Definition 16, exactly when there is a set of at least two consecutive suffixes addressed by r such that all the addressed suffixes have the same maximal common prefix, which is longer than that of any two other two suffixes, one taken in the set and one outside the set (note that any pair outside the set will have a common prefix of the same size that is necessarily different to the one we are considering). So, there is a minimum i , $1 \leq i < n$, and there is a number of occurrences $k \geq 2$ such that

- $u = w[r[i]..r[i] + |u| - 1]$,
- $LCP[i] = LCP[i + 1] = \dots = LCP[i + k - 2]$,
- $(LCP[i] > LCP[i - 1] \text{ or } i = 1)$ and $(LCP[i] > LCP[i + k - 1] \text{ or } i + k - 1 = n - 1)$. \square

Also, if a substring of w is not supermaximal to the left, there are at least two extensions by one symbol to the left that are equal.

Proposition 18. *A substring u in w is supermaximal to the left if and only if it occurs at least twice but at most $|\mathcal{A}|$ times in w and all its extensions by one symbol to the left are pairwise different.*

Proof. Supermaximality to the left requires that the previous symbol of each of the addressed suffixes be pairwise different. If any two of these suffixes had the same previous symbol, then there would be an extension to the left that is repeated twice. \square

Proposition 19. *A substring of a given string is a supermaximal repeat if and only if it is supermaximal to the right and to the left.*

The algorithm **findsmxr** first identifies the candidates that are supermaximal to the right by finding the set of consecutive positions in r that have local maximums in LCP . Then it filters out the candidates that are not supermaximal to the left. To check supermaximality to the left, we construct the set of previous symbols (symbols that occur immediately before each suffix on the set). Since the universe of this set is the size of the alphabet \mathcal{A} , it can easily be efficiently implemented in a boolean array of the size of \mathcal{A} . We take as an extra parameter an integer ml which is the minimum length of a supermaximal repeat to be reported (it can be set to 1 if desired). The pseudocode is given in Algorithm 4.

4.1 Correctness of findsmxr

Theorem 20 (Correctness). *The algorithm **findsmxr** computes all supermaximal repeats in the input string.*

Proof. Consider Algorithm 4. The main loop accesses sequentially the array LCP . By Proposition 19 supermaximal repeats are exactly the candidates that are supermaximal to the right and to the left. Propositions 17 and 18, characterize these properties with conditions on the data structures LCP and r . The algorithm checks exactly these conditions. \square

4.2 Complexity of findsmxr

Theorem 21 (Time complexity). *Given the suffix array for an input string of length n , **findsmxr** computes all supermaximal repeats in time $\mathcal{O}(n)$.*

Proof. The main loop iterates $n - 2$ times. Inside the main loop, all steps are clearly done in $\mathcal{O}(1)$ except the construction of the set on the innermost if clause. This is done by inserting each element into the set, represented as a boolean array, and maintaining the size appropriately. Each insertion takes $\mathcal{O}(1)$. Thus, the total number of insertions over all constructions of this set is the sum of the number of suffixes in each local maximum. Since no suffix can be in two different local maximums, this total is less than n . \square

Theorem 22 (Space complexity). *For an input of size n , algorithm **findsmxr** uses $\mathcal{O}(n)$ memory space. More precisely, it uses $n(2 \text{ word_size} + \log |\mathcal{A}| + 2) + |\mathcal{A}| + \mathcal{O}(1)$ bits.*

Proof. Consider Algorithm 4. The whole input w is allocated in memory. Since it contains n symbols, its memory usage is $n \log |\mathcal{A}|$ bits. The data structures r and LCP are arrays of length n whose elements are between 0 and n , and fit within the processor word. The described array for the set A has $|\mathcal{A}|$ bits. As before, since $|\mathcal{A}| \leq n$, the term $\log |\mathcal{A}|$ is considered $\mathcal{O}(1)$, as is the term word_size . \square

5 Implementation and experimental results

We implemented **findmxr** and **findsmxr** in C (ANSI C99), for a 32 or 64 bits machine.

For **findmxr** the memory space requirement is $n(3 \text{ word_size} + \log |\mathcal{A}| + 2)$ bits for an input of n . So, for \mathcal{A} the ASCII code and storing indices in 32 bits variables this becomes a total memory requirement of $13.25n$ bytes. In a 64 bits processor and 8 Gb RAM installed, the tool runs inputs of size up to ~ 618 Mb, without any swapping. Somewhat larger inputs can also be run efficiently because some swapping does not affect the running time. Notice

Table 1: Input data set used for comparison.

File	Description	Size (bytes)
linux	The Linux Kernel 2.6.31 tar file	365 711 360
HS-ch1	Homo-sapiens chromosome 1, from NCBI 36.49	251 370 600
ecoli	The file E.coli of the large Canterbury corpus	4 638 690
bible	The file bible.txt of the large Canterbury corpus	4 047 392
world	The file world192.txt of the large Canterbury corpus	2 473 400
a2M	The letter a repeated two million times.	2 000 000

Canterbury corpus can be found at <http://corpus.canterbury.ac.nz/descriptions/#cantrbry>, while the FASTA files for the NCBI 36.49 DNA human genome are downloadable at ftp://ftp.ensembl.org/pub/release-49/fasta/homo_sapiens/dna/

Table 2: Input size in bytes and running time in seconds of the three processes

File	Size	SA	findmaxr	findsmxr
linux	365 711 360	671.785	275.834	66.587
HS-ch1	251 370 600	798.583	197.545	100.714
ecoli	4 638 690	4.586	2.543	1.636
bible	4 047 392	3.201	2.085	0.750
world	2 473 400	1.670	1.224	0.377
a2M	2 000 000	2.423	29.135	0.111

that the in main loop of the Algorithm 3, the array I is only used one element at a time, so it can be easily handled by swap memory.

For **findsmxr** the memory space requirement is $n(2 \text{ word.size} + \log |\mathcal{A}| + 2) + |\mathcal{A}|$. Again, for \mathcal{A} the ASCII code and indices in 32 bits, the memory requirement is $9.25n$ bytes, which allows inputs of $\sim 885Mb$ in the same configuration.

We tested this implementation on large inputs, using an Intel® Core™2 Duo E6300 (only one core), running at 1.86GHz with 8GB RAM (DDR2-800) under Ubuntu linux for 64 bits.

The programs were compiled with the GCC compiler version 4.2.4, with option -O2 for normal optimization. The reported times are user times, counting *only* the time consumed by the algorithm, not including the time to load the input from disk. The input files used are described in Table 1. The files are chosen to demonstrate the behavior of the program for different kinds of natural data as well as degenerated cases. We used input sizes of the order of magnitude of the calculated limit, validating the performance in those cases. See [1] for experiments that push this to the actual limit.

Table 2 reports, for each case, the input size expressed in bytes and the running time expressed in seconds of three processes: *SA* our implementation of the suffix array construction of [14], *findmaxr* the computation of all maximal repeats with $ml = 1$, and *findsmxr* the computation of all supermaximal repeats with $ml = 1$. The time of both algorithms does not include the time for the construction of the suffix array. To know the time to compute the repeats from scratch, simply add both times.

5.1 Some remarks

The first step in our algorithms is the computation of the suffix array of the input string. We chose the smart algorithm of [14] that, profiting from the fact that the sorting is done on suffixes and not on arbitrary strings, it achieves a fast $\mathcal{O}(n \log n)$ time and requires $\mathcal{O}(n \log n)$ memory if the bit representation of each integer is accounted for.

An information-theoretic argument tells that $\Omega(n \log |\mathcal{A}|)$ bits are required to represent the permutation given by the suffix array, because there are $|\mathcal{A}|^n$ different strings of length n over the alphabet \mathcal{A} ; hence, there are at most that many different suffix arrays, and we need $\Omega(n \log |\mathcal{A}|)$ bits to distinguish between them. With suffix arrays $n \log n$ bits are used for this instead. *Compressed suffix arrays* were used to get closer to the lower bound ([3, 5, 13]), but at the cost of increasing the computational time [2] (for instance, when mapping substrings with many occurrences).

The algorithm of Lippert [15] is based on a compressed suffix array and finds all repeats of a *given length* in the input string. In contrast to our algorithm, Lippert's does not indicate whether a repeat is maximal. His experiments show that his solution requires much more time than ours.

The problem of finding repeats in large inputs occurs in genomic sequence analysis. A list of the most popular repeat finders for genomic sequences appears in the survey [19]. Leaving aside heuristic and library based methods such as RepeatMasker (2009), existing methods based on the suffix array still use the length of repeats as a parameter at each pass. The algorithm of [17] constructs a suffix array after building first a suffix tree of the input, hence yielding a very poor performance in terms of time and memory. A popular tool is REPuter, [12, 11]. It allows a very limited input size and its memory requirement depends on the repeat length and the number of occurrences in the input. In the worst case inputs are limited to RAM size/45. Since the output given by REPuter is not factorized, it becomes very large, needing $\mathcal{O}(n^2)$ space for inputs of size n .

References

- [1] Verónica Becher, Alejandro Deymonnaz, and Pablo Ariel Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. *Bioinformatics*, 25(14):1746–1753, 2009.
- [2] Alejandro Deymonnaz. Arreglos de sufijos para alineamiento de secuencias de ADN con memoria acotada. Technical report, Argentina, 2012.
- [3] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 397–406, New York, NY, USA, 2000. ACM.
- [4] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [5] Wing-Kai Hon, Tak Wah Lam, Kunihiro Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.

- [6] Bing Chao Huang. An algorithm for inverting a permutation. *Information Processing letters*, 12(5):237–238, 1981.
- [7] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal ACM*, 53(6):918–936, 2006.
- [8] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc.12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, London, UK, 2001. Springer-Verlag.
- [9] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley Professional, 1997.
- [10] M. Oguzhan Kulekci, Jeffrey Scott Vitter, and Bojian Xu. Efficient maximal repeat finding using the burrows-wheeler transform and wavelet tree. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 99(PrePrints), 2011.
- [11] Stefan Kurtz, J.V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, 2001.
- [12] Stefan Kurtz and Chris Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999.
- [13] T.W. Lam, K. Sadakane, W.K. Sung, and S.M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Lecture Notes in Computer Science. Computing and Combinatorics. 8th Annual International Conference COCOON 2002*, volume 2387, pages 401–410. Springer-Verlag. Heidelberg, 2002.
- [14] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.
- [15] Ross Lippert. Space-efficient whole genome comparisons with Burrows Wheeler transforms. *Journal of Computational Biology*, 12(4):407–415, 2005.
- [16] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. SODA '90: Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 319–327, San Francisco, 1990.
- [17] A. Poddar, N. Chandra, M. Ganapathiraju, K. Sekar, J. Klein-Seetharaman, R. Reddy, and N. Balakrishnan. Evolutionary insights from suffix array-based genome sequence analysis. *Journal of Biosciences*, 32(5):871–881, 2007.
- [18] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.
- [19] Surya Sahal, Susan Bridges, Zenaida V. Magbanua, and Daniel G. Peterson. Computational approaches and tools used in identification of dispersed repetitive dna sequences. *Journal Tropical Plant Biology*, 1(1):85–96, 2008.
- [20] A. F.A. Smit, R. Hubley, and P. Green. Repeatmasker, open-3.0. 1996-2004, 2009. <http://repeatmasker.org>.

This figure "time_vs_n_l.png" is available in "png" format from:

<http://arxiv.org/ps/1304.0528v1>

This figure "time_vs_n_p.png" is available in "png" format from:

<http://arxiv.org/ps/1304.0528v1>